Министерство науки и высшего образования Российской Федерации

Санкт-Петербургский Политехнический Университет Петра Великого

—

Институт компьютерных наук и кибербезопасности

ЛАБОРАТОРНАЯ РАБОТА № 1

«Лексический и синтаксический анализатор выражений»

1. по дисциплине «Формальные грамматики и теория компиляторов»
2. Выполнил
3. студент гр. 5151004/10101 Веремейчук Я.Ю.

<подпись>

1. Проверил
2. преподаватель Грибков Н.А.

<подпись>

1. Санкт-Петербург
2. 2024
3. Цель работы

Знакомство с Flex и Bison – лексическим и синтаксическим анализаторами.

1. Постановка задачи

Необходимо создать программу для разбора файлов с исходным кодом, написанным на C-подобном языке. Язык включает следующие конструкции:

1. Конструкция if-else.
2. Цикл while.
3. Операторы return, print, присваивание.
4. Операнды: десятичные числа, переменные (имя переменной начинается с латинской заглавной/строчной буквы; имя состоит из заглавных/строчных латинских букв, цифр, символа «\_»).
5. Арифметические операции: +, –, \*, /, %, унарный +, унарный –.
6. Логические операции: &&, ||, !.
7. Скобки (, ), обозначающие порядок выполнения операций.

Синтаксис отмеченных конструкций можно определить самостоятельно (например, использовать ли скобки с оператором print и т.д.). Синтаксис объявления переменных реализовывать не нужно.

Программа должна иметь функционал вывода ошибок: необходимо выводить номер строки и лексему, при разборе которой возникла ошибка.

Таким образом, программа принимает на вход файл с кодом на описанном выше языке. На выходе – сообщение о том, что разбор завершен успешно, или сообщение об ошибке, если код некорректный.

1. Подготовка к работе

3.1 Настройка окружения

Работа выполнялась на WSL2 с установленным дистрибутивом Ubuntu 22.04.2 LTS.

Перед началом работы необходимо скачать утилиту flex:

sudo apt update

sudo apt install flex

Чтобы запустить утилиту, которая принимает файл с исходным кодом программы на языке Lex (обычно с расширением .lex), необходимо ввести:

lex file.lex

Следующая команда компилирует сгенерированный файл на языке C (.yy.c). В результате выполнения этой команды будет создан исполняемый файл (a.out).

cc lex.yy.c

Запуск скомпилированной программы:

./a.out

3.2 ДКА

Для изучения механизма компиляции был рассмотрен пример из статьи (<https://habr.com/ru/articles/99162/>). В статье был дан код для написания калькулятора. Программа реализована в виде парсера, который при обработке символов руководствуется прописанными правилами.

ДКА (детерминированный конечный автомат) – сущность, которая может находиться в одном из N состояний; которая читает из входного потока символ за символом; и у которой есть таблица: для каждого сочетания текущего состояния и прочитанного символа – в которое состояние перейти. Работа flex заключается в том, что он строит ДКА по заданному набору регэкспов; в некоторых состояниях этот ДКА не только переходит в следующее, но вдобавок вызывает прописанные правила.

Увидеть построенный ДКА можно, заглянув внутрь сгенерированного парсера lex.yy.c. Для экономии места таблица переходов хранится не в явном виде, а разбитая на накладывающиеся списки. Чтобы увидеть её в наиболее наглядной форме, можно скомпилировать парсер с опцией -Cef («отключить сжатие таблиц»).

lex -Cef file.lex

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 1 – Таблица состояний для калькулятора-парсера

Отдельный массив static yyconst int yy\_ec[256] ставит каждому символу в соответствие класс.

Положительные числа в таблице переходов означают «перейти в состояние и продолжить читать»; отрицательные — «перейти в состояние и выполнить действие». Номер действия, которое должно выполняться по приходу в состояние, хранится в yy\_accept.

3.3 Автомат с магазинной памятью

Но ДКА не может распознавать вложенные конструкции из-за того, что у него недостаточно памяти: всё, что он помнит, — одно лишь значение от 1 до N (номер текущего состояния). Потребуется более ёмкий автомат, память которого сможет неограниченно расти по мере необходимости.

У автомата с магазинной памятью, кроме текущего состояния, есть стек, куда он может записывать символы. На каждом шаге автомат читает следующий входной символ, плюс верхний символ из стека. В соответствии с тройкой (текущее состояние, входной символ, символ из стека) автомат выбирает, в какое состояние перейти, и что записать в стек вместо прочитанного символа. Запись в стек необязательна: можно просто стереть оттуда прочитанный символ; так что стек во время работы может и расти, и сокращаться.

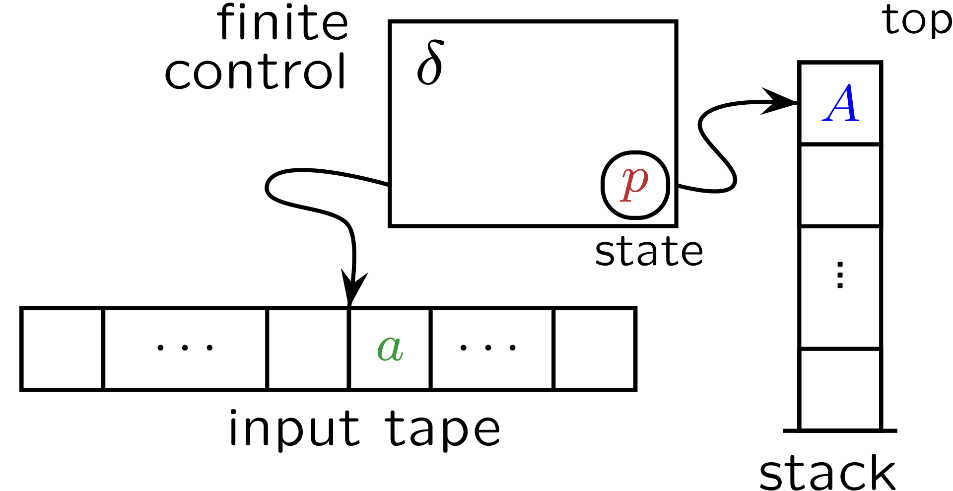


Рисунок 2 – Диаграмма автомата с магазинной памятью

Кроме ёмкого автомата, также потребуется новый способ задания синтаксиса — более гибкий, чем регэкспы.

3.4 Формальные грамматики

Для определения языковых конструкций, поддерживающих произвольную вложенность, стали использовать «правила вывода», состоящие из символов («терминалов») и переменных («нетерминалов»).

3.5 LR-парсинг

Магазинный автомат будет читать входную строку символ за символом, и записывать (сдвигать) прочитанные символы в стек. Как только наверху стека соберётся последовательность (символов и переменных), подходящая к правилу грамматики, автомат вытолкнет всю её из стека, и заменит на переменную, стоящую в левой части подошедшего правила (свёртка). Вся работа автомата заключается в последовательности сдвигов и свёрток.

3.6 Компиляция грамматики

Утилита bison может помочь создать парсящий автомат.

sudo apt-get install bison

Общий принцип такой же, как с flex: необходимо перечислить правила грамматики, и напротив каждого правила написать код на C, который будет выполняться во время свёртки правила.

Так же, как в lex-файлах, код в тегах %{ %} копируется в парсер неизменным. Есть две функции, которые необходимо определить: int yylex() возвращает следующий входной символ, и void yyerror(char \*s) печатает сообщение об ошибке. Классический yacc включал готовую реализацию yyerror, которую можно было в случае необходимости переобъявить; но его GNU-клон bison требует от программиста явную реализацию.

При компиляции парсера нужно указать библиотеку liby, в которой лежит стандартная бизонья main().

bison 2.y

cc -ly 2.tab.c

./a.out

При выполнении команды cc -ly 2.tab.c возникала ошибка.

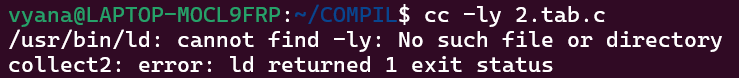


Рисунок 3 – Библиотека liby не была установлена вместе с пакетом bison

Решение проблемы – команда

sudo apt-get install libbison-dev

Бизоний парсер понимает строго заданный синтаксис. Чтобы предусмотреть дополнительные случаи, пришлось бы добавлять их в грамматику явно.

Итак, необходимо совместить преимущества flex и bison.

3.7 Двухступенчатый парсер

Можно было бы прогонять входной текст сначала через flex, а затем через bison, но нет необходимости получать промежуточный результат, обе ступени можно выполнить за один проход. flex читает входной текст символ за символом, время от времени передавая «наверх» бизону токены — терминальные символы грамматики. Значения для токенов flex задаёт сам.

Чтобы такой симбиоз был возможен, flex должен как-то узнать терминальные символы бизоньей грамматики. Для этого необходимо запускать bison с ключом -d, тогда он сгенерирует .h-файл с перечислением всех терминалов. Файл с определениями токенов получает суффикс .tab.h.

Компиляция двухступенчатого парсера:

lex 3.lex

bison -d 3.y

cc -ly lex.yy.c 3.tab.c

Достоинство бизона и ему подобных в том, что от усложнения языка вырастут в парсере только таблицы и свитч с действиями при свёртке, скопированными из .y-файла.

Весь остальной код парсера универсален: никаких спагетти, никаких рекурсивных функций, вызывающих друг друга в хитрых комбинациях. Правила грамматики действительно компилируются, а не обрамляются в синтаксис языка высокого уровня.

1. Ход работы

С помощью лексического анализатора lex и синтаксического анализатора bison был написан парсер кода на C.

Утилита lex с помощью регулярных выражений находит все необходимые лексемы (токены), заданные в файле «.lex». Для каждого оператора определен токен для дальнейшей обработки с помощью Bison. Если во входном потоке встречается цифра, то возвращается токен NUM, если название переменной, то – ID. Название переменной может начинаться только на латинские строчные/заглавные буквы. Также определены перевод строки и табуляция. Затем определяются символы операций, знаков и скобок, которые не имеют отдельного токена и передаются Bison в явном виде.

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Рисунок 4 – Распознаваемые токены

Необходимо помнить про CR и LF. Если парсер запускается на Unix-системе, то и в файлы должны быть созданы под эту систему. При работе с WSL возникает желание поработать с возможностями использования инструментов хостовой машины, например блокнота. Необходимо следить за настройками файла, иначе при запуске на Linux файла CR LF возникнет ошибка нераспознанного символа.



Рисунок 5 – Некорректный символ

Далее была написана грамматика, учитывающая приоритет сворачивания скобок, математических операций (\*, /, %, +, -), а также специфику конструкции if-else и цикла while. При написании грамматики описываются все операции, которые могут происходить.

Токены, которые были распознаны, подаются на вход утилите bison, которая строит по ним абстрактное синтаксическое дерево. В начале файла определяются токены, которые может возвращать lex.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 6 – Поддерживаемые токены

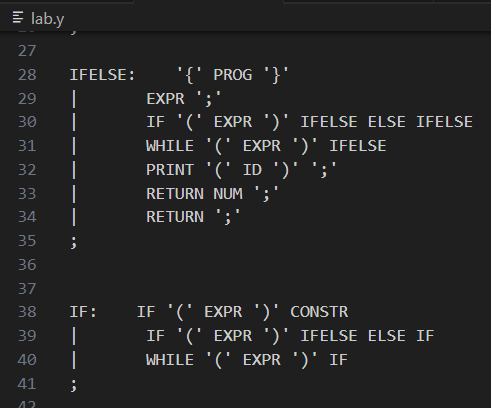


Рисунок 7 – Фрагмент грамматики

Также в функции main() было реализовано считывание информации из входящих данных.

Чтобы запустить парсер, необходимы команды:

lex lab.lex

bison -d lab.y

c++ lex.yy.c lab.tab.c

./a.out correct.txt

Изображение выглядит как текст, Шрифт, снимок экрана, Графика

Автоматически созданное описание

Рисунок 8 – Недостаточно входных аргументов

Изображение выглядит как текст, Шрифт, снимок экрана, Графика

Автоматически созданное описание

Рисунок 9 – Файл не найден



Рисунок 10 – Успешное завершение программы

Для определения строки с ошибкой в грамматике была переопределена функция «yyerror», которая выводит номер строки с ошибкой.

Для примера был рассмотрен корректный код на С.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 11 – Успешная проверка корректного файла

И некорректного.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 12 – Успешная проверка некорректного файла

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 13 – Успешная проверка некорректного файла

1. Выводы

В ходе работы были изучены лексический и синтаксический анализаторы – Flex и Bison, а также создана программа для разбора файлов с исходным кодом, написанным на С-подобном языке.

ПРИЛОЖЕНИЕ А

Листинг программы lab.lex

%{

#include <string>

#define YYSTYPE std::string

#include "lab.tab.h"

void yyerror(const char \*s);

void yysuccess(const char \*s);

%}

%option yylineno

%option noyywrap

%%

if return IF;

else return ELSE;

while return WHILE;

print return PRINT;

return return RETURN;

== return EQ;

[<]= return LE;

>= return GE;

!= return NE;

&& return LA;

[|][|] return LO;

[0-9]+ { yylval = yytext;

return NUM;

}

[a-zA-Z][a-zA-Z0-9\_]\* { yylval = yytext;

return ID;

}

[ \t\n] ; // whitespace

[-{};()=<>+\*/%!,] return \*yytext;

. yyerror("Error: Invalid character.");

%%

ПРИЛОЖЕНИЕ Б

Листинг программы lab.y

%{

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

extern FILE\* yyin; // File with code

extern int yylineno;

extern int yylex();

void yyerror(const char \*s) {

std::cout << "ERROR: " << s << ", line " << yylineno << std::endl;

exit(1);

}

void yysuccess(const char \*s) {

std::cout << s << std::endl;

}

#define YYSTYPE std::string

%}

%token IF ELSE WHILE PRINT RETURN

%token EQ LE GE NE LA LO

%token NUM ID

%%

PROG: CONSTR

| PROG CONSTR

;

IFELSE: '{' PROG '}'

| EXPR ';'

| IF '(' EXPR ')' IFELSE ELSE IFELSE

| WHILE '(' EXPR ')' IFELSE

| PRINT '(' ID ')' ';'

| RETURN NUM ';'

| RETURN ';'

;

IFNO: IF '(' EXPR ')' CONSTR

| IF '(' EXPR ')' IFELSE ELSE IFNO

| WHILE '(' EXPR ')' IFNO

;

CONSTR: IFELSE | IFNO ;

EXPR: CALC

| ID '=' EXPR

CALC: SUM

| CALC EQ SUM

| CALC LE SUM

| CALC GE SUM

| CALC NE SUM

| CALC LA CALC

| CALC LO CALC

| CALC '>' SUM

| CALC '<' SUM

;

SUM: MULT

| SUM '+' MULT

| SUM '-' MULT

;

MULT: VAL

| MULT '\*' VAL

| MULT '/' VAL

| MULT '%' VAL

;

VAL: NUM

| '-' VAL

| '!' VAL

| '(' EXPR ')'

| ID

;

%%

int main(int argc, char\* argv[]) {

if (argc != 2) {

std::cout << "ERROR: Input data mismatch." << std::endl << "Try: " << argv[0] << " <file.txt>" << std::endl;

exit(-1);

}

yyin = fopen(argv[1], "r");

if (yyin == NULL) {

std::cout << "ERROR: File was not found." << std::endl;

exit(-1);

}

yyparse(); // Start syntactic analyzer

yysuccess("OK: The program has been successfully completed.");

fclose(yyin);

return 0;

}